下载APP

课堂答疑（二）| C 工程实战篇问题集锦

2022-02-21 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 07:21 大小 6.74M

你好，我是于航。

在这门课的第三个模块“C 工程实战篇”中，我带你学习了在大型工程实战中应用 C 语言时需要掌握的很多必备技巧。而这次的答疑加餐，我从这个模块的课后思考题中精选了同学们讨论比较多，也比较有代表性的三个问题，来对它们进行详细分析。接下来，就请跟随我的脚步，一起来看看吧。

领资料

问题一

我在 14 讲 的“使用条件变量”一节中为你留下了这样一个问题：

在下面的代码中，为什么我们要在 while 语句，而不是 if 语句中使用 cnd_wait 呢？

复制代码

```
// ...
while (done == 0) {
  cnd_wait(&cond, &mutex);
}
// ...
```

对于这个问题，评论区的一些同学给出了不错的回答。比如 @liu_liu 同学就指出了其中的一个原因：

> 当阻塞的线程被重新调度运行时，done 的值可能被改变了，不是预期值。

还有 @ZR2021 同学也提到了与此相关的另一个重要因素：

> 使用 while 是防止广播通知方式的虚假唤醒，需要用户进一步判断。

这里，我就在同学们的回答基础上，对这个问题做一个总结。

首先需要知道的是，使用 while 或 if 语句的主要目的，在于判断线程是否满足“可以进入阻塞状态”的基本条件。比如，在上述代码中，当全局变量 done 的值为 0 时，表明当前线程需要优先等待其他线程完成某项任务后，才能够继续执行。但在现实情况中，“等待线程”的执行恢复往往会在各种非正常情况下发生。通常来说，这些情况可以被总结为三类。

第一种情况：在某些特殊的操作系统中，为了满足实现上的灵活性，**条件变量中已进入阻塞状态的线程，允许在未经 cnd_signal 函数通知的情况下被唤醒**。而在这种情况下，当前的程序状态可能并不满足被唤醒线程继续执行的条件。因此，使用 while 循环重新对条件变量进行检查，便成为了保证程序能够正确运行的重点。

第二种情况：在某些多处理器核心的系统上，**cnd_signal 函数的调用可能会同时唤醒所有等待线程**。因此，在“生产者与消费者”等类似场景中，让唤醒线程重新对条件进行检查，便能够防止因“产物不足”而导致的程序运行问题。

第三种情况：**由于线程之间存在竞态条件，可能会导致线程在被唤醒时无法正常处理需求。**

假设存在三个线程 A、B、C。其中，A、B 负责处理队列中的元素，而线程 C 负责向队列中生产元素。首先，A 线程获取元素，队列变为空；接下来，由于无元素可处理，B 线程进入阻塞状态。此时，C 线程生产了一个元素，放入队列，并唤醒一个阻塞线程（这里即 B 线程）。而当 B 线程正准备去处理这个元素时，该元素可能已经被 A 线程处理完毕。因此，让 B 线程在真正开始处理元素前，首先对条件进行重新检测便十分有必要了。

对于上述这三种情况，我们一般称其为线程的“虚假唤醒（Spurious Weakup）”。而通过使用 while 语句代替 if 语句，来在线程被唤醒时优先对条件进行重复检测，便可解决这个问题。如果想了解更多信息，你可以参考这个链接。

问题二

在 18 讲 中，我为你留了一个小作业，希望你去尝试了解一下什么是高速缓存的“抖动”。这里我来具体解释一下。

因为程序设计正好与运行所在平台的高速缓存策略（包括物理特性）产生冲突，导致同一个高速缓存块被反复加载和驱逐，这个过程就是“高速缓存抖动（Thrashing）”。在这种情况下，每一次加载到高速缓存中的数据都没有被程序充分利用，因此，程序的整体运行效率会有明显的下降。

我们可以通过 CSAPP 中的一段代码来理解缓存抖动的发生过程。这段代码如下所示：

复制代码

```
float foo(float x[8], float y[8]) {
  float sum = 0.0;
  int i;
  for (i = 0; i < 8; i++)
    sum += x[i] * y[i];
  return sum;
}
```

假设一台计算机拥有两个高速缓存行，每个缓存行大小为 16 字节，且它们被分别组织在不同的两个组中。对于这样的缓存行分组方式，我们一般称其为“直接映射高速缓存”。在这种情况下，上述代码在每一次交替访问数组 x 与 y 中的元素时，便可能会产生高速缓存抖动的问题。

在直接映射高速缓存中，数据在缓存中的存储位置是与它在内存中的地址一一映射的。在上面这个例子中，我们将所使用数据地址的最后 4 位作为“块偏移”的索引，用于指定所需数据在某个缓存行中的偏移位置（以“字节”为单位）；而地址的倒数第 5 位将用于选择不同的组。假设数组 x 与 y 在内存中被连续存放，`float` 类型为 4 字节大小，且 x 的起始地址从 0 开始。那么，我们可以得到如下表所示的数据与高速缓存行映射关系：

元素	地址（后五位）	组索引	元素	地址（后五位）	组索引
x[0]	b'**0**0000	0	y[0]	b'**0**0000	0
x[1]	b'**0**0100	0	y[1]	b'**0**0001	0
x[2]	b'**0**1000	0	y[2]	b'**0**1000	0
x[3]	b'**0**1100	0	y[3]	b'**0**1100	0
x[4]	b'**1**0000	1	y[4]	b'**1**0000	1
x[5]	b'**1**0100	1	y[5]	b'**1**0100	1
x[6]	b'**1**1000	1	y[6]	b'**1**1000	1
x[7]	b'**1**1100	1	y[7]	b'**1**1100	1

极客时间

可以看到，数组 x 与 y 中相同索引位置上的元素会被缓存到同一个缓存行中（同一个组）。因此，当交替访问它们时，高速缓存中的数据便会被不断地加载和驱逐，缓存并没有得到有效利用。

当然，对于上面这个例子，我们可以简单地通过将 x 定义为包含有 12 个元素的数组（额外填充 16 字节，即对应一个缓存行大小），来将两个数组相同索引位置上的元素“拆分”到不同的缓存行中。但总的来看，缓存抖动在采用了“直接映射高速缓存”的体系中较为常见，而在采用“全相联高速缓存”及其他缓存策略的体系中较少发生。因此，一般来说，我们并不需要对代码进行任何处理来预防这类问题。但当程序性能真正出现问题时，你可以将“高速缓存抖动”这个原因考虑进去。

问题三

在 19 讲 的思考题中，我提到了一个名为“达夫设备”的概念。那么它有什么作用？它的实现原理是怎样的呢？

达夫设备（Duff’ s Device）是 C 语言发展历史中出现的一个**利用手动循环展开进行性能优化**的知名实践案例，它由 Tom Duff 于 1983 年发现。

相较于我在 19 讲中介绍的循环展开方式，达夫设备通过结合使用 switch 语句与 do-while 语句，使得循环的“余数次项”可以不用单独进行处理。它的原始形式主要用于将 16 位的无符号整数从一个数组中复制到 MMIO 寄存器。而经过改写，我们可以得到以下更“普适”的版本。这个版本的代码可用于将指定内存段上的数据复制到另一个内存段中：

复制代码

```
#include <stdio.h>
void send(int* from, int* to, int count) {
  int n = (count + 7) / 8;
  switch (count % 8) {
    case 0: do { *to++ = *from++;
    case 7:      *to++ = *from++;
    case 6:      *to++ = *from++;
    case 5:      *to++ = *from++;
    case 4:      *to++ = *from++;
    case 3:      *to++ = *from++;
    case 2:      *to++ = *from++;
    case 1:      *to++ = *from++;
                } while (--n > 0);
  }
}
int main(void) {
  int x[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  int y[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
  send(x, y, 10);
  for (int i = 0; i < 10; i++) {
    printf("%d", y[i]);
  }
}
```

可以看到，达夫设备的实现代码并不是正常程序员会写出来的（笑）。它利用了 case 分支的 “fall-through” ，即可以连续执行的特性，使得 switch-case 与 do-while 语句可以像上述这样被结合起来使用。

这里程序在执行时，switch 语句会首先根据总迭代次数在当前循环展开形式下（这里即 8x1）的“余数次项” ，跳转到对应分支（这里即 “case 2” ）开始执行。而当程序按顺序执行完 “case 1” 对应的分支语句后，紧接着会对 while 语句的循环条件进行判断。这

里，变量 n 用于控制 while 循环的执行次数。因此，若该值在递减后仍大于 0，则程序会重新从 “case 0” 语句，即 do...while 循环的起始处，开始新一轮的执行。

达夫设备采用的循环展开方式可以**减少不直接有助于程序结果的操作数量**，如循环索引计算与条件分支，因此可以在一定程度上提升性能。

但需要注意的是，通常来说，编译器在高优化等级下会自动对符合条件的代码采用循环展开。并且，从可读性、稳定性，甚至执行性能等角度考虑，上述用来进行数组元素拷贝的达夫设备代码，也并不如 C 标准库提供的 memcpy 函数来得有效。

好了，以上就是我对这门课第三个模块中，同学们讨论比较多的三个课后思考题的回答。如果你还有其他问题，欢迎在评论区与我讨论，或者进入课程专属的微信群（进群入口），与其他小伙伴一起沟通交流。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 课堂答疑（一）| 前置篇、C 核心语法实现篇问题集锦

下一篇 LMOS说C语言 | 为什么说C语言是一把瑞士军刀？

更多学习推荐

2021 最新大厂
Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

限量免费领取

精选留言 (3)

写留言

Ping
2022-02-21

多谢老师

作者回复: 不客气哈！

1

ALPHA
2022-02-23

老师好，问一哈c语言有类似于协程的特性吗

作者回复: C 语言本身没有协程哈，一般都是用 setjmp 和 longjmp 来模拟协程的特殊运行方式。

ZR2021
2022-02-21

666